webserv: Building a Non-Blocking Web Server in C++98 (A 42 project)

MannBell · Follow

5 min read · Jan 14, 2024

In the vast realm of internet protocols, the Hypertext Transfer Protocol (HTTP) stands as the backbone of communication for the World Wide Web. To truly comprehend the intricacies of web communication, 42 introduced to us a wonderful project to embark, that challenges our skills and understanding as always… Yes we are talking about the "Webserv" project, which involves creating a custom HTTP server in C++98, with a particular emphasis on non-blocking behavior.

Project Overview:

— The project at its core is about building a server in C++98.
Why C++98? I believe that 42 purposefully does that, so that students have a deeper understanding of how C++ is implemented(and its history).

— It must be a non-blocking server.
Building a web server in C++ involves various challenges, and one common issue is related to blocking operations. In the context of web servers, blocking refers to the situation where a thread is waiting for an operation to complete before it can proceed further. This can lead to performance bottlenecks, especially in scenarios where the server needs to handle **multiple concurrent connections**.

The most significant blocking issue arises ***when dealing with input/output (I/O) operations, such as reading from or writing to sockets***. In a traditional blocking model, a thread will wait until data is available or until a write operation is completed. During this waiting period, the thread is essentially inactive, and the server may struggle to handle other incoming requests.

To address this, we will employ asynchronous I/O; Asynchronous I/O allows a server to initiate an I/O operation and continue with other tasks while waiting for the operation to complete. This way, a single thread can manage multiple connections simultaneously without being blocked.

Asynchronous I/O can be done by using various available system calls like `select` , `poll` , `epoll` (Linux) ...

`poll` and `epoll` are simpler/better alternatives to `select` , but `select` is more portable (can be used accross different platforms), so let's talk about it a bit:

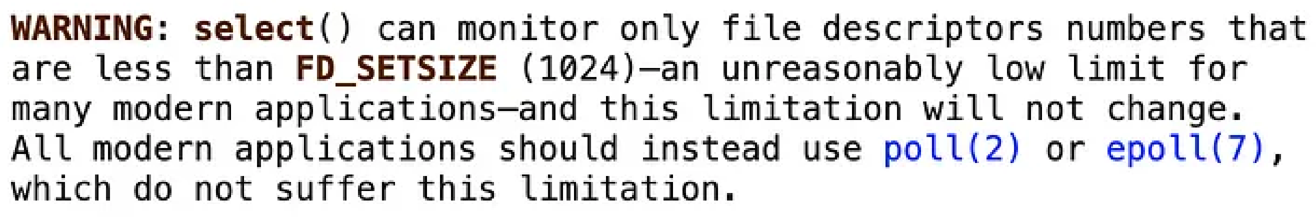
`select` is a system call in Unix-like operating systems that facilitates the asynchronous monitoring of multiple file descriptors for specific events, such as read, write, or errors. It enables a program to efficiently wait for and handle events on a set of descriptors. Here's an overview of how `select` works:

1. **Initialization of fd_set:**
   We initialize separate fd_set structures for each type of event(read or write...) we wish to monitor. These structures serve as bit masks representing sets of file

descriptors(*so that we can know which FDs are ready for read/write, bit 1 for ready* and 0 for not).

2. **Populating the fd_set:**
   File descriptors of interest are added to the respective fd_set using macros like `FD_SET`.
3. **Setting the Timeout:**
   We can set a timeout value, to determine the maximum duration `select` should wait for an event. If the timeout elapses without any events, `select` returns, allowing the program to proceed.
4. **Calling `select`:**
   The program invokes the `select` function, providing the `"highest file descriptor value" + 1` (*man select*), the `three fd_sets`, and the `timeout`; We can set `NULL` as value for events that we are not interested in, or for `timeout` if we wish `select` to block indefinitely until an event occurs.
5. **Blocking or Returning:**
   `select` enters a blocking state, awaiting events on the monitored file descriptors or until the timeout expires. Upon an event or timeout, `select` returns control to the program.
6. **Checking the fd_set:**
   After returning, the program examines the fd_sets to identify which file descriptors are ready for the specified events, by using macros like `FD_ISSET`.
7. **Handling Events:**
   We can respond to events by executing the required actions based on the identified file descriptors. For instance, we may read/write data from/to a socket or handle errors.
8. **Looping or Exiting:**
   We often employ a loop to repeat the monitoring process, allowing continuous asynchronous handling of events on multiple file descriptors.

While `select` is known for its portability across different platforms, it has limitations, such as a maximum number of file descriptors it can handle. That's why modern alternatives like `poll` and `epoll` are recommended instead, as the manual states:



```
WARNING: select() can monitor only file descriptors numbers that
are less than FD_SETSIZE (1024)—an unreasonably low limit for
many modern applications—and this limitation will not change.
All modern applications should instead use poll(2) or epoll(7),
which do not suffer this limitation.
```

https://man7.org/linux/man-pages/man2/select.2.html

— Implementing the HTTP/1.1 .

After setting up the server for basic reading and writing operations, we can now move on to parsing what we exchange with the clients(i.e. messages).

A typical HTTP communication is about comprehending the requests and generating appropriate responses, and to do so, it is crucial to understand the structure of HTTP messages.

I will go lightly over HTTP/1.1 here, but in case you are looking for more depth, i advise you to read the official RFCs, It's essential to focus on the latest versions — *I invested a considerable amount of time reading rfc2616, only to realize it is obseleted very long ago*; as of writing this article, the current ones are rfc9110 and rfc9112.

HTTP messages consist of a request or response line, headers, an empty line(`CRLF` or `\r\n`), and an optional message body.

Here is an illustration using Augmented Backus–Naur Form:

```
HTTP-message   = start-line CRLF
                   *( field-line CRLF )
                   CRLF
                   [ message-body ]

start-line     = request-line / status-line
```

1. **Request Line:**

```
request-line   = method SP request-target SP HTTP-version
```

The request line in an HTTP request message contains the method, URI, and HTTP version. Parsing this line is the first step in understanding the client's request.
For example:

```
GET /path/to/resource HTTP/1.1
```

- In the above example, "GET" is the method, "/path/to/resource" is the URI, and "HTTP/1.1" is the version.

2. **Status Line/Response Line**

```
 status-line = HTTP-version SP status-code SP [ reason-phrase ]
```

Similarly, in an HTTP response message, the status/response line contains the HTTP version, status code, and a reason phrase. Extracting this information is crucial for generating meaningful responses.
For example:

```
HTTP/1.1 200 OK
```

- Here, "HTTP/1.1" is the version, "200" is the status code, and "OK" is the reason phrase(*or the description of the status code*).

3. **Headers:**

Both requests and responses can include headers, providing additional information about the message. Parsing headers involves extracting key-value pairs to

understand various aspects such as content type, content length, and more.
For example:

```
Host: example.com
Content-Type: text/html
Content-Length: 256
```

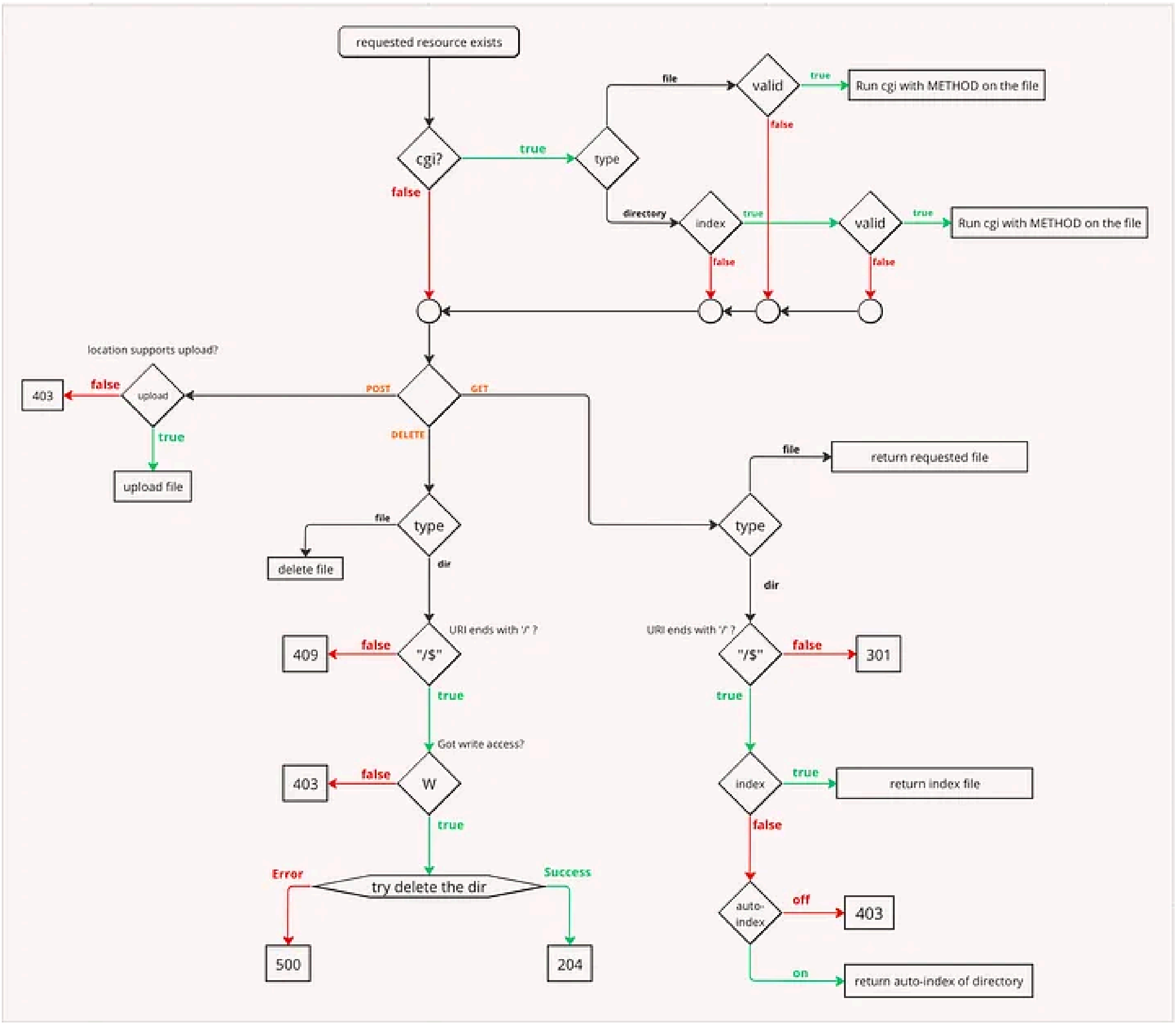
Parsing these headers enables the server to interpret the content appropriately.

4. **Message Body:**

The message body, though optional, may contain data relevant to the request or response. Parsing the message body depends on factors like content type and length, transfer coding etc...

After finishing parsing the HTTP messages, the next steps involve implementing logic to handle different HTTP methods, process requests, and generate appropriate responses.

Here is an example of a flow-chart I created, to have an overview on the reponse process:



Note: It may not be exhaustive or flawless; its purpose is to provide a straightforward overview rather than encompassing all the details

That’s it folks! I hope you found the article somewhat useful, I aimed to provide an overview of the project and spotlight certain aspects that I found beneficial. Thank you sincerely for taking the time to read, and I wish you a wonderful day! :)

You can always check my github at:
https://github.com/m4nnb3ll

Socket Programming · Http Request · Web Server · 42 Network · Cpp